LR115906

(2)

UNLIMITED

DTIC FILE COPY

RSRE
MEMORANDUM No. 4438

AD-A231 319

ROYAL SIGNALS & RADAR ESTABLISHMENT

NODEN USER'S GUIDE & INSTALLATION MANUAL

Author: C H Pygott

PROCUREMENT EXECUTIVE,
MINISTRY OF DEFENCE,
RSRE MALVERN,
WORCS.

91 1 29 106

RSRE MEMORANDUM No. 4438

UNLIMITED

ROYAL SIGNALS AND RADAR ESTABLISHMENT

Memo 4438

TITLE: NODEN USER'S GUIDE & INSTALLATION MANUAL

Compiler version 3.x
Analyser version 7.x
Comparison version 7.x

AUTHOR: C H Pygott

DATE: November 1990

SUMMARY

This document is intended to provide the NODEN user with a guide to the facilities of the tools and an explanation of some of the error messages that may be generated.

The annexes of this memo provide a guide to the installation of the NODEN tools on DEC VAX machines (under VMS) and SUN 3 workstations.

Contents

1 Introduction and basic operation

This section will describe the basis operation of the NODEN tool set. The detailed operation of the compiler, analyser and comparison programs will be described in later sections. The installation of these tools are described in the appendices.

In all examples in this memo it is assumed that the advice in the installation guide has been followed and that the compiler, analyser and comparison programs have the logical names 'noden', 'analysis' and 'compare' respectively.

The aim of the NODEN tool set is to show that an implementation meets its specification under all circumstances. That is, the user starts by specifying how the outputs of a device depend upon the state of its inputs[1], and also how the device is to be implemented as a network of gates. The NODEN tools then attempt to show that for all input conditions the implemented behaviour is compatible with that specified.

This means that the user starts with two files in the NODEN Hardware Description Language (NODEN_HDL [Pygott#1]): one for the specification and the other for the implementation. Whilst the two files have the same root name, the specification must have the extension '.noden' and the implementation '.cnoden'.

A simple example of the use of NODEN is given in appendix C.1. The specification says that the output *op* is the exclusive OR function of the two inputs *a* and *b*. This can be analysed [Pygott#2] to show that:-

$$op = (\bar{a} \cdot b) + (a \cdot \bar{b})$$

It can also be shown that the implementation's output *xor* depends upon its inputs *x* and *y* in the same way. This enables the comparison program to report that the implementation is correct.

If the specification and implementation files are called 'example.noden' and 'example.cnoden' respectively, then as implied above, comparing the specification and implementation of a device requires five operations:-

- Compiling the specification to check the legality of the source file and generating the input for the analyser program
- Similarly, compiling the implementation
- Analysing the compiled specification and generating a file that contains the required behaviour

[1]for devices with memory, the current state of the memory is regarded as an additional input to the device, and the next state of the memory an additional output.

- Similarly, analysing the compiled implementation to generate a file containing the implemented behaviour
- Comparing the specified and implemented behaviours and reporting agreement or any errors

These are performed by the commands:-

```
noden    example s            (Compile EXAMPLE spec)
noden    example c            (Compile EXAMPLE circuit)
analysis example s            (analyse EXAMPLE spec)
analysis example c            (analyse EXAMPLE circuit)
compare  example              (compare behaviours)
```

The comparison program at this point will either report that the implementation is valid, or will report the circumstances under which the implementation and specification disagree. This report is sent to the screen, and can also be found in the file 'example.ancom'.

The following sections will describe the operation of these three programs in more detail, and indicate some of the possible error messages. More complex and complete examples of the use of NODEN can be found in Appendix C.

2 The source files

As illustrated in the introduction, the user starts with two descriptions in NODEN_HDL. The language is described in the NODEN_HDL manual and familiarity with the language is assumed in the rest of this memo. However, there are some aspects of the source files that are not covered by the language manual, and these are discussed here.

2.1 The units to be compared

As is described in the NODEN_HDL manual, NODEN descriptions are formed from three function types: 'FN's which are auxiliary functions, 'BLOCK's to be analysed, and 'CIRCUIT's which are collections of blocks.[2].

When comparing specification and implementation, the BLOCKs are the units which are to be compared. That is the specification file should contain one or more BLOCK descriptions, and the implementation file should contain BLOCKs that represent the implementations of those requirements.

The comparison program checks that the analysed specification and implementation files contain the same number of BLOCKs (and reports an error if they don't). If they do have the same number of BLOCKs, the comparison program assumes that corresponding

[2]CIRCUITs are largely ignored by the current tools, but exist to describe how the BLOCKs are joined together for simulation, or for future more abstract verification work.

BLOCKs in the two files occur in the same order, ie the first BLOCKs from both files are compared, then the second etc.

When comparing BLOCKs the comparison program also assumes that comparable BLOCKs will have the same signature, that is the same input and output types, and the same internal memories[3]. This may involve the user having to add an association block to the implementation, to make the implemented form of the requirement look like the specified form (see the NODEN_HDL manual).

2.2 Differences between specification and implementation files

Logically, there is only one difference in the analysis of the specification and implementation files. The function of the analysis program is, for each BLOCK, to calculate how each combinatorial output and next memory state are derived from the combinatorial inputs and current memory states. It is possible that during this evaluation some set of states is identified for which the result is undefined. If the NODEN description being analysed is a specification, then this is reported as an error, whilst it is not an error for an implementation (as the specification may say it doesn't care what the implementation delivers under those circumstances). That is the specification must be defined under all circumstances.

This is most likely to arise when the optional ELSE clause is omitted from an IF or CASE statement. For example, a three state type could be defined as follows, together with a function that is only defined for two of the states:-

```
TYPE t3 = WIRE (m1 | m2 | m3).

FN M1_OR_M2 = (t3: a) -> bool: CASE a OF  m1: t,  m2: f  ESAC.
```

In the following block B1, the output isn't defined when *a* is *m3*, whilst B2 is defined under all circumstances.

```
BLOCK B1 = (t3: a) -> (bool: op): M1_OR_M2 a.

BLOCK B2 = (t3: a) -> (bool: op):
     IF a /= m3  THEN  M1_OR_M2 a  ELSE f FI.
```

As a specification, B1 will be reported as erroneous during analysis, whilst as an implementation both blocks will analyse correctly. The error message during the analysis of B1 as a specification will effectively say "BLOCK B1 output 'op' is undefined when input 'a' = m3".

In practice, there tends to be a different style of description in the specification and implementation files. The specification tends to be more abstract in both its use of enumeration and integer types, and its use of comparatively complex operations such as addition and subtraction. Specification descriptions tend to define the required behaviour in a function style, with much use of conditional statements.

[3]Note that whilst the types must be the same, the names can be different.

By contrast, an implementation description tends to be in three parts. The first is a 'library' of simple functions that represent the primitive gate elements from which the implementation is to be made (such as NAND gates etc). The second part describes how the implementation is constructed from a network of these primitive gates (using MAKEs and JOINs). The third part consists of association blocks that show how the implementation corresponds to the specified BLOCKs. In particular, how inputs and outputs are ordered, how the implemented boolean signals map onto the more abstract types used in the specification etc.

3 The 'noden' command

The introduction shows the compiler being used to generate the analyser inputs for the specification and implementation. Whilst these are the normal operations of the compiler, it can also be used in a number of other modes.

3.1 'noden's parameters

The command 'noden' can be used with none, two or three parameters. If used with no parameters, a brief summary of the parameter options is displayed as on-line help.

If the command has parameters, the first is always the name of the file (without extension) to be compiled. The second compulsory parameter says what function the compiler should perform, and the third (optional) parameter indicates any diagnostic or other 'unusual' requirements. The compiler functions are:-

- Second parameter SPEC (or S), generate the analyser input for the specification (in *file*.anil[4], see Figure 1)
- Second parameter CIRCUIT (or C), generate the analyser input for the implementation (in *file*.ancil)
- Second parameter ELLA (or E), translate the specification file into ELLA [Morison] for simulation (in *file*.elt)
- Second parameter C_ELLA, translate the implementation file into ELLA for simulation (in *file*.elt)
- Second parameter SYNTAX, perform syntax and type checks on the specification file, but don't generate analyser input or ELLA files
- Second parameter C_SYNTAX, perform syntax and type checks on the implementation file, but don't generate analyser input or ELLA files

[4]where *file* is the value of the first parameter.

The case of the second parameter is ignored (ie may be upper or lower case, but not mixed). Specification source text is always taken from *file*.noden, whilst implementation source is in *file*.cnoden.

All uses of the compiler generate a diagnostic file, *file*.syout. This contains a copy of all error messages displayed on the screen, and any diagnostic information requested by use of the optional third parameter.

If ELLA output is generated, it is in *file*.elt, and includes all FNs, BLOCKs and CIRCUITs in the source text. In order to simulate the NODEN_HDL description, one of the ELLA primitive files NODEN_BOOL.ELT or NODEN_NO_BOOL.ELT[5] must first be compiled into an ELLA context. The ELLA model generated uses the same names as the NODEN_HDL description for types, type members and FNs, BLOCKs and CIRCUITs (all of which become ELLA FNs). A few points to note are:-

- NODEN wire and enumeration types become ELLA enumeration types with the same name and members, but with two extra members 'x_x*name*' and 'x_x*name*'. These are the don't care and undefined members respectively. That is:-

```
NODEN:-      TYPE bool = WIRE (t | f).
becomes
ELLA:-       TYPE bool = NEW (t | f | x__xbool | x__ubool).
```

- A NODEN integer type becomes an ELLA integer type, with type name 'x_i*name*' and tag '*name*'. Its range is also increased by two, to provide the don't care and undefined states (in that order). That is:-

```
NODEN:-      TYPE int3 = INT[0..7].
becomes
ELLA:-       TYPE x__iint3 = NEW int3/(0..9).
```

- The ELLA generated from the NODEN description has been flattened so that function definitions are not nested. This means that functions defined inside other NODEN functions appear in the ELLA listing with an additional integer added to the name, eg '*name*_1'. This differentiates it from other possible functions with the same name declared in other scopes. That is, the NODEN_HDL description:-

```
FN F1 = ................
  FN INNER = ...........
  FN IN1   = ...........
  BEGIN ................
    END.

FN F2 = ................
  FN INNER = ...........
  FN IN2   = ...........
  BEGIN ................
    END.
```

[5]depending upon whether the NODEN_HDL description uses the in-built boolean type.

becomes the ELLA:-

```
FN INNER__1 = ...........
FN IN1__2   = ...........
FN F1       = ...........
FN INNER__3 = ...........
FN IN2__4   = ...........
FN F2       = ...........
```

3.2 Third parameter options

The third parameter takes the form of an integer in the range 0 to 512.[6] This is formed as the sum of the following options[7]:-

0 or 1:	No partial evaluation required (see below)
2:	As each block is read from the source, the user will be asked if partial evaluation of that block is required (reply 'y' or 'n')
3:	Partial evaluation is assumed to be required for all blocks
Above +4:	The amount of data space used by internal data objects is displayed at the end of the compilation.
Above +8:	The compiler's name table is printed in the diagnostic file in an undocumented format. This option is not recommended!
Above +16:	A map of the abstract representation of the source is printed in the diagnostic file in an undocumented format. This option is not recommended!
Above +32:	A map of a flattened abstract representation of the source is printed in the diagnostic file in an undocumented format. This option is not recommended!
Above +64:	No effect
Above +128:	Suppress the compiler's warning messages when it performs type coercion
Above +256:	Produce a listing in the diagnostic file of each lexical unit as it is read from the source file. This option is basically not recommended, but may be useful as a last resort if the compiler is reporting a syntax error which the user cannot understand.

Hence, "`noden example s 7`" compiles the specification of some device (from 'example.noden') and causes partial evaluation of all blocks (3) & data space usage to be displayed (4).

[6] If omitted, its default value is 0, ie no additional diagnostics.
[7] Note that some of the options are there to assist in program development and fault diagnosis, and so are not recommended for the general user.

Any block can be partially evaluated. That is, its behaviour can be calculated on the assumption that some or all of its inputs are in fixed states. Only simple inputs can be fixed, ie single wire, enumeration or integer values, not array objects. However, any structures are regarded as flattened to simple and array values, so simple values in structures can be fixed. For each input which may be fixed the user is asked to provide its value. This must be either a value of the correct type, or a null return (which is taken to mean that the value should be taken as a variable, with all possible values being considered). For example, if partial evaluation is to be applied to a block with inputs defined by:-

```
BLOCK B1 = (bool: a, [2]bool: b, (bool, [2]bool, t3): c) .......
```

then the user will be asked for the value of a, `c[1]` & `c[3]`.

This facility is useful when considering descriptions too large to analyse in one attempt. For example, an ALU could be analysed to check that it adds correctly, then separately that it subtracts correctly etc. The comparison program checks that the blocks in the specification and implementation have been analysed with the same fixed values. Note that the state of memory elements cannot be fixed.

3.3 Compiler error messages

Most compiler error messages should be straight forward and self explanatory. Some of the more likely ones are:-

- File not found. The '.noden' or '.cnoden' file doesn't exist.
- Syntax errors. In the case of syntax errors, a list of acceptable symbols at that point in the text is provided. Note that some in-built types and functions start with the character '_'. These are reported as being of the lexical type of the character which follows. That is, '_bool' is said to be a 'lower case name' and '_AND' an 'upper case name'.
- Type errors. If an attempt is made to use functions with operands of the wrong type, to have different typed objects in separate limbs of an IF or CASE statement or to deliver a value of the wrong type from a function, a type error message will be generated.
- Bad enumeration type definitions. When enumeration types are generated with defined boolean representations, it is illegal to have unused bits in the representation, for example:-

  ```
  TYPE ok = NEW word3 (o1 = #000 | o2 = #0x1 | o3 = #1x0 ).

  TYPE no = NEW word3 (n1 = #0x0 | n2 = #0x1 | n3 = #1x0 ).
  ```

  The second example is illegal because the second bit of the representation is never used. It is also illegal to define the representation of some members but not of others.

- Bad CASE selectors. Errors in selector values (such as being of the wrong type) are reported when the selector is used. An error is reported at the end of a CASE statement if the same selector is used twice.

- Bad MAKEs and JOINs. All functions made with a MAKE statement, must have their inputs defined in a JOIN statement. An error is reported if a function is made but never joined, or if an attempt is made to define the inputs of a function twice.

- Bad association block. In an association block, a set of statements are used to define how an implementation's memory elements map on to those of the specification. Each memory element in the implementation must be referenced once and once only.

- Description is not acyclic. NODEN descriptions may not contain any delayless loops. If they do, an error is reported in the form of a list of user function names that form the loop. Each function is described as a path name in the form of a list of names separated by ':'s. For example: 'A:b:C<2>'. This is read as the second occurrence of the user function 'C' which is called within a function named locally as 'b', which in turn is called from the first (or only) occurrence of a function 'A'. Note that the test for delayless loops is only performed if the compiler is generating an analyser input file (second command parameter SPEC or CIRCUIT).

  For example, the following is a slightly odd description of what is effectively a cross coupled pair of NAND gates:-

```
FN INV = (bool: a)   -> bool: NOT a.
FN AD  = (bool: a b) -> bool: a AND b.

BLOCK B1 = (bool: a b) -> (bool: op):
     BEGIN MAKE AD: a1 a2.
           JOIN (a, INV a2) -> a1,
                (b, INV a1) -> a2.
           OUTPUT a1
       END.
```

This leads to the error message:-

```
Function 'B1' is not acyclic

   Output 'op' is driven from a loop formed by :-

     a1
     INV<1>
     a2
     INV<2>
     a1
```

Note that the list starts and ends at the same function, and that the message could equally have reported an error in the generation of 'a2', but having listed one loop it suppresses other 'linked' loops.

The compiler may possibly (though hopefully unlikely) fail with a message "FATAL ERROR: ......". Unless the message is immediately meaningful, this probably is an error in the program (one of its internal consistency checks has failed). The error and the code that produced it should be reported to the program supplier. This also applies to the analyser and comparison programs.

4 The 'analysis' command

The introduction has illustrated the two uses of the analyser; to calculate the behaviour described by the specification and implementation from their respective compiled source texts. However, as with the compiler, there are a number of additional diagnostic options that will be described here. Also, the analyser's error messages are potentially more obscure than the compiler's, and these will also be described.

4.1 'analysis' parameters

Like the compiler, the command 'analysis' can be used with none, two or three parameters. If used with no parameters, a brief summary of the parameter options is displayed as on-line help.

If the command has parameters, the first is always the name of the file to be analysed and the second compulsory parameter says whether the specification or implementation is being analysed. The third (optional) parameter indicates any diagnostic or other 'unusual' requirements.

If the second parameter is "SPEC" (or "S"), the analyser takes the specification analyser input (from *file*.anil[8]) and calculates the specified behaviour (in *file*.anres).

Similarly, if the second parameter is "CIRCUIT" (or "C"), the implementation analyser input (from *file*.ancil) is used to generate the implemented behaviour (in *file*.acirc).

Both uses of the analyser generate a diagnostic file, *file*.syout. This contains a copy of all error messages displayed on the screen, and any diagnostic information requested by use of the optional third parameter.

4.2 Third parameter options

The third parameter takes the form of an integer in the range 0 to 63.[9] This is formed as the sum of the following options:-

0: No diagnostics required

[8]where *file* is the value of the first parameter.
[9]If omitted, its default value is 0, ie no additional diagnostics.

Above +1:	Display the time taken to analyse each block, and the total amount of memory used by internal data objects. Note that the time taken is described as CPU time. This is true on the VAX, but on the SUN it is actually the elapsed time
Above +2:	Display the name of each block as its analysis starts
Above +4:	At the start of each block the user is asked if diagnostics are required (answer 'y' or 'n'). If the answer is yes, the inputs and outputs of the block and all operations performed by the analyser are displayed. This is not recommended!
Above +8:	As above, but diagnostics are provided for all blocks.
Above +16:	The values calculated by the analyser are printed in the diagnostic file in an undocumented format. This is not recommended!
Above +32:	As above in a different undocumented format.

Hence "`analysis example c 17`" analyses the implementation (from 'example.ancil'). It displays the resources used during analysis (1) and prints a diagnostic form of the result in the diagnostic file (16).

4.3 Analyser error messages

Assuming the analyser can find the appropriate input file ('.anil' or '.ancil'), it should normally run with no problems and produce a message "All blocks analysed correctly". There are however two circumstances under which errors will be reported by the analyser.

If the description being analysed is a specification. as has already been said in section 2.2, all the values calculated must be fully defined. That is, there shouldn't be any circumstances under which an output has the 'undefined' value. If such a circumstance is found, an error report is given (similar to that to be shown shortly). Such an erroneous block is marked in the output file as 'unavailable' and if examined by the comparison program will lead to the block automatically being reported as incorrect. Should the same source text be analysed as an implementation, no error will be reported[10].

The second class of errors can arise when mapping values from the integer type to a row of booleans (with WORD*n* or S_WORD*n*) or to a constrained integer type. In all these cases the integer type can have more values than are expressible in the target type. This can be illustrated as follows:-

```
TYPE int   = INT[0..5].
MAP  I_MAP = _integer -> int.

BLOCK B1 = (bool: inc, int: a) -> (int: b):
      IF inc THEN I_MAP(a + 1) ELSE a FI.

BLOCK B2 = (bool: inc, int: a) -> (int: b):
      IF inc THEN  IF a == 5 THEN 5 ELSE I_MAP(a + 1) FI  ELSE a FI.
```

[10]though the output will still be undefined under the appropriate circumstances. This may lead to the comparison program reporting an error, if the specification says that the output should be significant.

BLOCK B1[11] attempts to add 1 to *a* when *inc* is true, or else passes *a* unaltered. However, when *a* has the value 5, the incremented value is not a member of the constrained integer type *int*. This leads to the following error message:-

```
FAIL in block B1

RANGE CHECK for type 'int' on line 5 of source applied to too large a value
under the following circumstances:-

                                        inc = t
                                          a = 5
```

Note that the fact that the problem only arises if *inc* is true is recognised by the analysis. BLOCK B2 is analysed correctly, as the potentially illegal increment is trapped and a suitable legal value delivered.

5 The 'compare' command

The introduction has illustrated the use of the comparison program to compare the calculated behaviours of the specification and implementation. As with the compiler and analyser, there are a number of additional diagnostic options that will be described here. The interpretation of the comparison program's reports will also be discussed.

The command 'compare' can be used with none, one or two parameters. If used with no parameters, a brief summary of the parameter options is displayed as on-line help.

If the command has parameters, the first is always the name of the files to be compared. The second (optional) parameter indicates any diagnostic or other 'unusual' requirements.

The analysed specification and implementation behaviours are taken from *file*.anres[12] and *file*.acirc respectively. The results of the comparison (together with any diagnostics) are written to *file*.ancom.

5.1 Second parameter options

The second parameter takes the form of an integer in the range 0 to 15.[13] This is formed as the sum of the following options:-

0: No diagnostics required

Above +1: Display the time taken to compare each block, and the total amount of memory used by internal data objects. Note that the time taken is described as CPU time. This is true on the VAX, but on the SUN it is actually the elapsed time

[11]the mapping function I_MAP is explicitly written in B1, as this is the operation that is going to fail, if it was not written there explicitly the compiler would put one in anyway to coerce the output to the correct type. The compiler also coerces *a* to integer before +.

[12]where *file* is the value of the first parameter.

[13]If omitted, its default value is 0, ie no additional diagnostics.

Above +2: For each block, list which combinatorial inputs and current memory values influence each combinatorial output and next memory state.

Above +4: Print a diagnostic trace of all inputs and outputs of each block, as seen by both specification and implementation. This is not recommended!

Above +8: Write the values read in by the comparison program to the diagnostic file in an undocumented format. This is not recommended!

Hence "`compare example 3`" compares the specified and implemented behaviours (in 'example.anres' and 'example.acirc'). It displays the resources used during comparison (1) and prints the dependencies of each output and memory element (2).

5.2 Comparison messages

The comparison program first checks that it can find the analyser output files (*file*.anres and *file*.acirc') and reports an error if it cannot. It then checks that the two files contain the same number of blocks. If they do, it then starts comparing the first blocks from both files, then the second etc.

For each block a check is made that the inputs, outputs and memory elements in the specified and implemented forms are comparable. That is whilst the specified and implemented versions need not have the same type names, if the first input of the specified block is an enumeration type with six members, the same must be true of the implemented version. At this stage a check is also made that any inputs that have been set with a defined value (for partial evaluation) have been evaluated with the same values in both specification and implementation.

If the specified or implemented behaviours of a block are unavailable, it is reported as being erroneous. A block's behaviour may be unavailable if it raised an analyser error as described in the previous section or if the body of the block or the body of any of the functions it uses are defined as being "EMPTY" (see the NODEN_HDL manual).

Assuming that the specified and implemented behaviours are compatible and available, the comparison program then compares the analysed behaviours of each combinatorial output and next state of the memory elements. The user may request a dependency list for each output. This is illustrated in the following example:-

```
[3]

BLOCK COUNTER = (bool: reset inc) -> (^word3: count):
      IF reset THEN 0
    ELIF   inc THEN IF count == 7 THEN 0 ELSE count + 1 FI
             ELSE count
      FI.
```

This is the specification of a block defining the behaviour of a three bit counter. The use of the '^' character before the output type name indicates that this is a memory element. The name *count* can therefore used in the body of the block to mean the current state of the counter, and the value of the expression that forms the body of the block is taken as the next state of *count*. The use of '~' in the output signature in this way causes the block to have a combinatorial output called *_count* in addition to the memory element called *count*. That is the above description is identical to the following NODEN_HDL block[14]

```
[3]

DELAY WD = word3.

BLOCK EQUIVALENT = (bool: reset inc) -> (word3: _count):
     BEGIN MAKE WD: count.
           LET next =    IF reset THEN 0
                        ELIF   inc THEN IF count == 7 THEN 0 ELSE count + 1 FI
                                   ELSE count
                          FI.
           JOIN next -> count.
           OUTPUT count
       END.
```

The dependency list from the comparison program is as follows:-

```
Specification: 'COUNTER'       Implementation: 'COUNTER'

Specification output '_count[1]' - implementation output '_count[1]' depends on
inputs:-
                                             count[1]

Specification output '_count[2]' - implementation output '_count[2]' depends on
inputs:-
                                             count[2]

Specification output '_count[3]' - implementation output '_count[3]' depends on
inputs:-
                                             count[3]

Specification output 'count[1]' - implementation output 'count[1]' depends on
inputs:-
                                               reset
                                                 inc
                                             count[1]

Specification output 'count[2]' - implementation output 'count[2]' depends on
inputs:-
                                               reset
                                                 inc
                                             count[1]
                                             count[2]
```

[14]except this block is illegal because of the leading '_' in the output name. This equivalence is explained in more detail in the NODEN_HDL manual.

```
Specification output 'count[3]' - implementation output 'count[3]' depends on
inputs:-
                                                  reset
                                                    inc
                                               count[1]
                                               count[2]
                                               count[3]
```

Note that the names of the outputs and memory elements as defined by both the specification and implementation are listed, but the names in the list of inputs is taken from the specification (in this case all the names happen to be identical).

Should an error be discovered, a message similar to that produced by the analyser is generated. For example if the above specification had been implemented wrongly, the following message may have been produced:-

```
COMPARISON ERROR: Implementation output 'count[2]' is incompatible with
the specification of 'count[2]' under the following circumstances:-

                                                  inc = t
                                                reset = f
                                             count[1] = t
```

Again the names are taken from the specification. Whilst these input conditions can steer the user to the general region of the problem, they are not necessarily sufficient to deliver an incorrect result. For example, a device may be specified as:-

```
BLOCK B1 = (bool: a b c) -> (bool: op): IF a THEN b ELSE c FI.
```

but implemented as:-

```
BLOCK B1 = (bool: a b c) -> (bool: op): IF a THEN b ELSE NOT c FI.
```

This would lead to a report from the comparison program saying they disagreed when "a = f".

The analyser may also give warnings if an output from the implementation seems to depend upon different inputs from the corresponding output in the specification. However, note that these are only warnings, as the specification may be over precise. For example, an output may be specified to have the value:-

```
          IF a THEN b ELSE bool FI
```

That is, the specification says that the output must be *b* when *a* is true, but at other times the implementation can deliver any value. This specification can correctly be implemented by always delivering *b*. This means that the specification depends on *a* and *b*, whilst the implementation only depends on *b*. This will be flagged as a warning by the comparison program, but the block will be reported as correct.

References

[Morison] J. Morison, N. Peeling & T. Thorp: 'ELLA: Hardware description or specification?', Proc IEEE international conference CAD-84, Santa Clara Nov 1984

[Pygott#1] C. Pygott: 'The NODEN hardware description language', RSRE Report 89011, August 1989

[Pygott#2] C. Pygott: 'The algebra of the NODEN analyser', RSRE Report 89012, August 1989

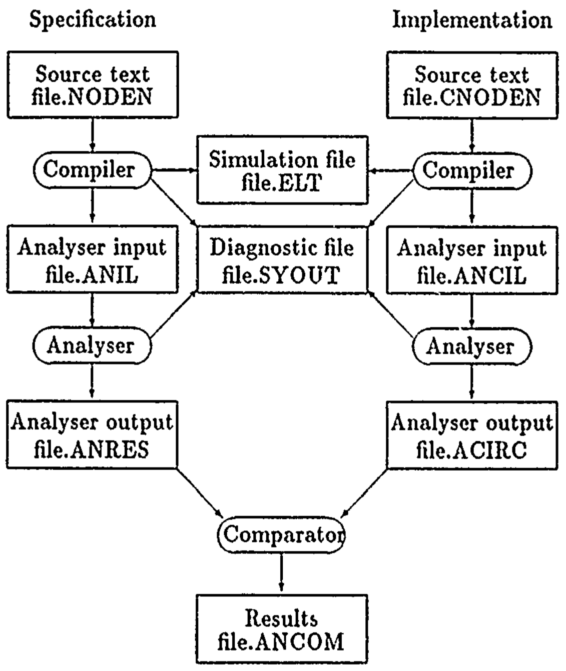

Figure 1: Use of the NODEN suite, with file name extensions

THIS PAGE IS LEFT BLANK INTENTIONALLY

A VMS installation guide

The NODEN tools set is distributed on half-inch magnetic tape. This should be read into a suitable directory, where it will be found to contain several subdirectories. A '.readme' file will indicate what files are on the tape and in particular what examples are included. The actual tools consist of three executable images and associated command macros, these are detailed below.

A.1 Compiler installation

The compiler is run by a macro 'noden_hdl.com'. This contains a system dependent line that must be altered for each installation. The line:-

```
$ RUN [....]COM_NODEN
```

must be altered, so that the path name leads to the directory containing the 'com_noden.exe' executable image.

It is recommended that the command 'noden' is defined in the user's 'login.com' file, as:-

```
$ noden :== "@[......]noden_hdl"
```

where the path name leads to the directory containing the 'noden_hdl.com' macro.

A.2 Analyser installation

The analyser is run by a macro 'analyser.com'. This contains a system dependent line that must be altered for each installation. The line:-

```
$ RUN [....]ANALYSER
```

must be altered, so that the path name leads to the directory containing the 'analyser.exe' executable image.

It is recommended that the command 'analysis' is defined in the user's 'login.com' file, as:-

```
$ analysis :== "@[......]analyser"
```

where the path name leads to the directory containing the 'analyser.com' macro.

There are two other lines in the 'analyser.com' macro that the user may wish to change.

When the analyser is run normally, all diagnostics and reports that are sent to the screen are also copied to the diagnostic file (with the '.syout' extension). If it is desired to run the analyser in batch mode, it is possible to suppress the screen outputs (which would appear in the '.log' file) by changing the line:-

```
        $  BATCHRUN = 0
to
        $  BATCHRUN = 1
```

Also, the analyser has a fixed size array for expressions it is holding internally. It is possible (though unlikely) that this array may become full, leading to an error message saying that 'the LET table is full'. The size of this table can be increased from its current value of 500, by changing the value of the integer in the line:-

```
$LP3:    LETTABLEMAX = 500
```

A.3 Comparison installation

The comparison program is run by a macro 'comparison.com'. This contains a system dependent line that must be altered for each installation. The line:-

```
$  RUN [....]COMPARISON
```

must be altered, so that the path name leads to the directory containing the 'comparison.exe' executable image.

It is recommended that the command 'compare' is defined in the user's 'login.com' file, as:-

```
$  compare :== "@[......]comparison"
```

where the path name leads to the directory containing the 'comparison.com' macro.

There is one other line in the 'comparison.com' macro that the user may wish to change. When the comparison program is run normally, all diagnostics and reports that are sent to the screen are also copied to the results file (with the '.ancom' extension). If it is desired to run the comparison program in batch mode, it is possible to suppress the screen outputs (which would appear in the '.log' file) by changing the line:-

```
        $  BATCHRUN = 0
to
        $  BATCHRUN = 1
```

B SUN installation guide

The NODEN tools set is distributed on a SUN cassette in 'tar' format. This should be read into a suitable directory, where it will be found to contain several subdirectories. A '.readme' file will indicate what files are on the tape and in particular what examples are included. The actual tools consist of three executable images and associated command macros, these are detailed below. It is assumed that the tools are to be run under SunView.

B.1 Compiler installation

The compiler is run by a c-shell called 'noden'. This contains a system dependent line that must be altered for each installation. The line:-

```
~/..../com_noden "DIAGLEVEL="$diaglevel .......
```

must be altered, so that the path name leads to the directory containing the 'com_noden' executable image.

It is recommended that the command 'noden' is defined in the user's '.cshrc' file, as:-

```
alias noden 'csf -f ~/....../noden'
```

where the path name leads to the directory containing the 'noden' c-shell.

B.2 Analyser installation

The analyser is run by a c-shell called 'analysis'. This contains a system dependent line that must be altered for each installation. The line:-

```
~/...../analyser "DIAGLEVEL="$diaglevel \
```

must be altered, so that the path name leads to the directory containing the 'analyser' executable image.

It is recommended that the command 'analysis' is defined in the user's '.cshrc' file, as:-

```
alias analysis 'csf -f ~/....../analysis'
```

where the path name leads to the directory containing the 'analysis' c-shell.

There are two other lines in the 'analysis' c-shell that the user may wish to change.

When the analyser is run normally, all diagnostics and reports that are sent to the screen are also copied to the diagnostic file (with the '.syout' extension). If it is desired to run the analyser in batch mode, it is possible to suppress the screen outputs by changing the line:-

```
                set batchrun = 0
     to
                set batchrun = 1
```

Also, the analyser has a fixed size array for expressions it is holding internally. It is possible (though unlikely) that this array may become full, leading to an error message saying that 'the LET table is full'. The size of this table can be increased from its current value of 500, by changing the value of the integer in the line:-

```
                set lettablemax=500
```

B.3 Comparison installation

The comparison program is run by a c-shell called 'compare'. This contains a system dependent line that must be altered for each installation. The line:-

```
                ~/...../comparison \
```

must be altered, so that the path name leads to the directory containing the 'comparison' executable image.

It is recommended that the command 'compare' is defined in the user's '.cshrc' file, as:-

```
                alias compare 'csf -f ~/....../compare'
```

where the path name leads to the directory containing the 'compare' c-shell.

There is one other line in the 'compare' c-shell that the user may wish to change. When the comparison program is run normally, all diagnostics and reports that are sent to the screen are also copied to the results file (with the '.ancom' extension). If it is desired to run the comparison program in batch mode, it is possible to suppress the screen outputs by changing the line:-

```
                set batchrun = 0
     to
                set batchrun = 1
```

C Example of the use of NODEN

This appendix will show the NODEN tools being used to verify the implementation of two simple requirements. The first is a combinatorial circuit, whilst the second is a circuit with memory elements. The clocking of the memory elements in the second example leads to some particular problems that will be discussed in the last two sections of this appendix.

C.1 Combinatorial example

The simplest use of NODEN is in the verification of combinatorial circuits. A very simple requirement may be:-

```
      [1]  \ Built-in boolean type required \

BLOCK XOR = (bool: a b) -> (bool: op): a /= b.
```

That is the specification defines the behaviour of an exclusive OR gate. This can be implemented by four NAND gates as shown in Figure 2. Textually, this circuit can be described as:-

```
      [1]

      FN NAND2 = (bool: a b) -> bool: a NAND b.

      BLOCK XOR_IMP = (bool: x y) -> (bool: xor):
            BEGIN MAKE NAND2: g1 g2 g3 g4.
                  JOIN ( x, y) -> g1,
                       ( x,g1) -> g2,
                       ( y,g1) -> g3,
                       (g2,g3) -> g4.
                  OUTPUT g4
          END.
```

The implementation description is in two parts, a description of the primitive gates used in the circuit[15] and a description of the network of primitive gates that form the circuit. Normally, it would be expected that this network would be a function, with an association block used to map the implemented form to the specified form (as will be seen in the other examples), but in this case the implementation is already in the correct form, so the association block isn't required.

If the above two descriptions are in files 'xor.noden' and 'xor.cnoden', then as illustrated in the introduction, the following five operations will confirm the correctness of the implementation:-

```
compile  xor s
compile  xor c
```

[15]although in this example the built-in NAND function could have been used directly.

```
analysis xor s
analysis xor c
compare  xor
```

C.2 Verification of functional correctness of clocked circuits

Verification of clocked circuits requires consideration to be given to the modelling of the clock. This is a concern because NODEN's model of delays assumes a purely synchronous design. That is, with all delays being updated by a global and implied 'clock'. In the actual circuit, there will be an explicit clock signal. Part of the verification process must therefore be to show that the actual clock causes the implemented delay elements to be updated at the same time as NODEN's model thinks they are updated. The next three sections will show three approaches to this problem.

The requirement for the device in all the following sections is simply that it acts as a three bit register. That is the output is the current state of the register and the next state is the value of the current input. This is specified in NODEN_HDL as:-

```
[3]

BLOCK LATCH = (word3: a) -> (^word3: op): a.
```

This is implemented by the circuit shown in Figure 3. This design has been selected to illustrate a number of points about clock distribution and possible differences between the implemented and specified forms of the device. In this case the implementation has some built-in test logic in the form of a limited scan-path (which was not described in the specification) and there is some logical manipulation of the clock signal.

The simplest approach to the verification of such a device is to assume that the clock distribution is correct and to consider the functional correctness of the implementation only. This is illustrated in this section. It should be noted that the assumptions made about the clock distribution affect the models of the primitive gates in the circuit and the association block, but do not affect the description of the circuit as a network of gates.

As was described in section 2.2, the circuit in Figure 3 can be described in NODEN_HDL in three parts: the primitive gates, the gate network and an association block:-

```
[3]

\ --------------------- Primitive gates ------------------------ \

FN INV  = (bool: a)   -> bool: NOT a.

FN AND2 = (bool: a b) -> bool: a AND b.

FN MUX  = (bool: sel a b) -> bool: IF sel THEN a ELSE b FI.

DELAY DB = bool.  \ a boolean delay element \

FN D_TYPE = (bool: data clock) -> bool: DB data.
```

The description of the combinatorial gates is straight forward, but the description of the D_TYPE needs some consideration. As has been said, it is to be assumed that the clock distribution is correct, so D_TYPE can ignore the *clock* input. Its behaviour is therefore identical to that of the NODEN's built-in DELAY function, latching the input *data* value on each implied clock tick.

```
\ -------------------- Circuit ----------------------------- \

FN IMP = (bool: run shift_in clock, word3: d) -> word3:
   BEGIN MAKE INV:  g1 g2 g3,
              AND2: g4,
              MUX:  g5 g6,
              D_TYPE: g7 g8 g9.
         JOIN g3 -> g1,
              g3 -> g2,
              clock -> g3,
              (run,clock) -> g4,
              (run,d[1],shift_in) -> g5,
              (run,d[2],g7) -> g6,
              (g5,g1) -> g7,
              (g6,g2) -> g8,
              (d[3],g4) -> g9.
         OUTPUT (g7, g8, g9)
   END.
```

This is a textual form of the circuit shown in Figure 3. Note that the circuit has three inputs in addition to data input defined in the specification. These are the actual clock and two signals relating to the scan-path.

The intended operation of the circuit is that when *run* is true the circuit behaves as specified, but when *run* is false it is in test-mode. When in test-mode, the first two memory elements form a two-bit shift register with *shift_in* as the shift register's input. The third memory element doesn't alter its state in test-mode.

```
\ -------------------- Association block ----------------------------- \

BLOCK LATCH_I = (word3: a) -> (word3: op):
   MAP MAKE IMP: imp.
       MAKE (word3: latch) = (g7:DB, g8:DB, g9:DB).
       JOIN (t, !bool, !bool, a) -> imp.
       OUTPUT imp
   END.
```

The final part of the implementation description is an association block to make the implementation appear to have the same form as the specification. That is it says:-

- which one-bit register elements in the implementation (and in which order) form the three-bit register in the specification. Each memory element in the implementation is identified by a path name that leads to a NODEN DELAY function. That is '`g7:DB`' is the DELAY function DB used in the function named *g7*, which by implication is in the function IMP

- how the specification's output corresponds to that of the implementation (in this case they are identical)
- how the implementation's inputs are derived from those of the specification

This latter point needs some expansion. It has already been said that only functional correctness during normal operation is to be considered. This means that *run* can be fixed as true, and the states of *shift_in* and *clock* can be ignored. Note that these two inputs are given the illegal boolean value, to check that they are indeed never significant (as if they are used, an undefined output will be generated).

The above specification and implementation can be compiled, analysed and compared as before, and the comparison program reports that the circuit is correct. What this means is that the implementation agrees with the specification, provided that the following caveats hold true:-

- The device is in normal running mode, not test-mode. This was a decision taken by the user, who hasn't said anything about test-mode.
- The hardware is working correctly. That is the gates are behaving as their models say they should. NODEN can say that the design of a device is correct, but not that a particular physical realisation of that design is correct. In practice, for critical systems, this may mean the use of two or more devices in parallel to protect against hardware failures.
- The clock frequency is sufficiently low. NODEN ignores time, other than the quantized time defined by the implied clock. In effect it assumes that all combinatorial functions are evaluated in zero time. In practice this isn't so, so at some high enough clock frequency the device will not perform as specified. Finding this frequency is the preserve of traditional CAD tools.
- The clock is correctly distributed to all the register elements. That is that all registers are updated at the same time. Often this simply means checking that all register elements are clocked from the same signal, but in circuits such as the above example where the clock is buffered or manipulated a more complex 'proof obligation' exists. The next two sections demonstrate two solutions to this problem.

C.3 Verification of the clock signal

One way of verifying the correctness of a clocked circuit is to verify the functional behaviour on the assumption that the clock is correctly distributed (as shown above), and then to redefine the boolean type so that it includes values representing rising and falling clock signals. If the primitive gate element definitions are suitably modified, it is possible to demonstrate that the correct clock edge (say rising) reaches all the register elements.

This can be done simply by using one description and the compiler & analyser. If the description is treated as a specification, and D_TYPEs are defined so as to produce an undefined next memory state unless the correct clock signal is received, the analyser will report an error if any of the D_TYPEs receives the wrong clock value.

The boolean types can be redefined as:-

```
[NO_BOOL 3]

\ --------------------- Redefine 'bool' ------------------------ \

TYPE bool = WIRE (rising | falling | t | f | glitch).
TYPE word3 = [3]bool.
```

That is, it includes true (rising) and inverted (falling) clocks and a *glitch* state which can be used as an illegal clock. The primitive gates can then be redefined as:-

```
\ --------------------- Primitive gates ------------------------ \

FN INV  = (bool: a)   -> bool:
   CASE a OF
       rising: falling,  falling: rising,  t: f,  f: t,  glitch: glitch
   ESAC.


FN AND2 = (bool: a b) -> bool:
   CASE a OF
       rising: CASE b OF rising: rising, t: rising, f: f ELSE glitch ESAC,
      falling: CASE b OF falling: falling, t: falling ELSE glitch ESAC,
            t: b,
            f: CASE b OF falling: glitch ELSE f ESAC,
       glitch: CASE b OF f: f ELSE glitch ESAC
   ESAC.


FN MUX  = (bool: sel a b) -> bool: glitch.
```

Note that as the MUX is not used in the clock circuit, no definition need be given for it. Hence it can always return an illegal clock value. The D_TYPE is redefined as:-

```
DELAY DB = bool.

FN D_TYPE = (bool: data clock) -> bool:
   DB (IF clock == rising THEN t ELSE !bool FI).
```

The D_TYPE is still simply a NODEN DELAY, but now its next value is either an arbitrary legal value if the correct clock signal is seen, or otherwise it is the undefined boolean.

The circuit is identical to the text shown in C.2. As the specification file must contain a BLOCK that uses the implementation, a dummy BLOCK function is written as:-

```
\ --------------------- Dummy block ------------------------------ \

BLOCK LATCH_I = (word3: a) -> (word3: op): IMP(t, !bool, rising, a).
```

Note that only behaviour during normal running (*run* is true) is being considered, so *shift_in* has no influence and is `!bool`. The clock input is now significant and is set to the normal (rising) value.

This description can be compiled and analysed, and the analyser reports that the block analysed correctly. This indicates that the clock signal correctly reaches all the D_TYPEs.

C.4 Verification of the scan-path

The previous two sections have shown NODEN verifying the behaviour of the circuit under normal operating conditions. However this could be extended to included a specification of the test-mode behaviour as well. The specification could then be expressed as:-

```
[NO_BOOL 3]

\ --------------------- Redefine 'bool' ------------------------ \

TYPE bool = WIRE (rising | falling | t | f | glitch).
TYPE word3 = [3]bool.

FN PASS = (bool: a) -> bool: IF (a == t) _OR (a == f) THEN a ELSE bool FI.

BLOCK LATCH = (word3: a, bool: run shift_in) -> (^word3: op):
      IF run == t
        THEN (PASS a[1], PASS a[2], PASS a[3])
    ELIF run == f
        THEN (PASS shift_in, PASS op[1], PASS op[3])
        ELSE word3
      FI.
```

Note that the boolean in the specification must be the same as that to be used in the implementation (ie must contain the clock values), but that the specification doesn't want to consider what happens if a clock is applied to (for example) the *shift_in* input. Hence the definition of the function PASS to ensure that the specification only defines the behaviour of the device when the non-clock inputs are true or false.

The implementation is again in three parts. The specification of the boolean type and the INV & AND2 gates is as given in section C.3. However now the MUX function is significant and must be defined, and the description of D_TYPE needs to show that if it gets a rising clock it latches the data input, but that if it receives no clock it doesn't change state:-

```
FN MUX  = (bool: sel a b) -> bool:
   IF sel == t THEN a ELIF sel == f THEN b ELSE glitch FI.

DELAY DB = bool.

FN D_TYPE = (bool: d clk) -> bool:
   BEGIN MAKE DB: q.
         LET  next_q = CASE clk OF rising: d, f: q ESAC.
         JOIN next_q -> q.
         OUTPUT q
     END.
```

The circuit network is again identical to that given in section C.2, but the association block is now:-

```
BLOCK LATCH_I = (word3: a, bool: run shift_in) -> (word3: op):
   MAP MAKE IMP: imp.
       MAKE (word3: latch) = (g7:q, g8:q, g9:q).
       JOIN (run, shift_in, rising, a) -> imp.
       OUTPUT imp
   END.
```

The compiler, analyser and comparison programs can again be used as shown in the introduction to verify the correctness of this implementation under both normal and test conditions. Note that the following caveats still hold:-

- The hardware is working correctly.
- The clock frequency is sufficiently low.
- The device is being used sychronously. That is the clock is applied to the clock input, and the data applied to the other inputs is stable over the period of the clock.

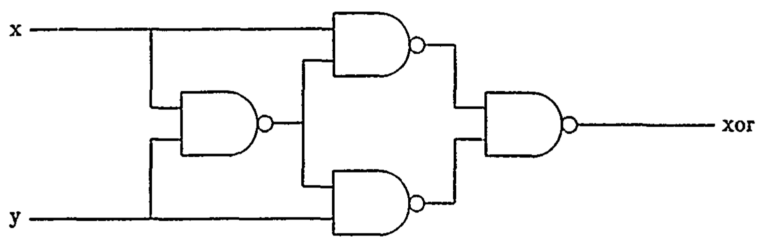

Figure 2: Possible implementation of combinatorial circuit

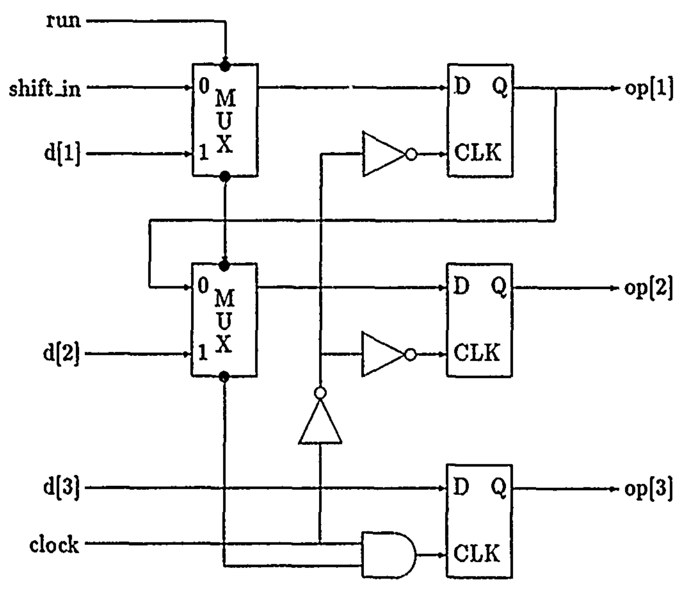

Figure 3: Possible implementation of clocked circuit

REPORT DOCUMENTATION PAGE

DRIC Reference Number (if known) ....

Overall security classification of sheet ....Unclassified..

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the field concerned must be marked to indicate the classification eg (R), (C) or (S).

Originators Reference/Report No. MEMO 4438	Month NOVEMBER — Year 1990
Originators Name and Location RSRE, St Andrews Road Malvern, Worcs WR14 3PS	
Monitoring Agency Name and Location	
Title NODEN USER'S GUIDE & INSTALLATION MANUAL	
Report Security Classification UNCLASSIFIED	Title Classification (U, R, C or S) U
Foreign Language Title (in the case of translations)	
Conference Details	
Agency Reference	Contract Number and Period
Project Number	Other References
Authors PYGOTT, C H	Pagination and Ref 30

Abstract

This document is intended to provide the NODEN user with a guide to the facilities of the tools and an explanation of some of the errcr messages that may be generated.

The annexes of this memo provide a guide to the installation of the NODEN tools on DEC VAX machines (under VMS) and SUN 3 workstations.

Abstract Classification (U,R,C or S) U

Descriptors

Distribution Statement (Enter any limitations on the distribution of the document)

UNLIMITED

S80/48